# Responses to reviewers' comments for Paper # 278

We are deeply grateful to all the reviewers for the insightful suggestions which help us to improve this work to a better level. In the following, we provide an explicit response to each raised concern.

## 1. RESPONSE TO REVIEWER 1

**1.1** *The main con that I've spotted is the fact that the algorithm might have been more nicely framed also within the Spark environment by taking advantage of the various possibilities offered by it (e.g. caching of RDDs)*

**Response:** We agree with the reviewer that the solutions could benefit from utilizing advanced Spark features. In our previous submission, our SPARE algorithm has already taken advantage of two distinct features of Spark, including DAG execution engine and caching of RDDs, to accelerate its performance:

**DAG Execution Engine:** As described in Section 5.3, our SPARE algorithm uses a run-time best-fit strategy to achieve load balance. This strategy relies on the DAG execution engine, which is a distinct feature of Spark. The strategy can be viewed as a planning stage in the Map-Planning-Reduce workflow. In particular, after the Map stage, we can collect the sizes of all stars as input for better workload balance among the reducers. Whereas, there is no such a planning stage in Hadoop.

**Caching of RDD:** We also utilize the RDD caching feature of Spark to store the map result (i.e., RDDs of stars) during the planning stage and reuse them in the reduce stage.

In our revision, we have edited Section 5.3 to explicitly clarify these points. Nevertheless, we believe there could exist other nice features of Spark worth further exploration in our solutions.

## 2. RESPONSE TO REVIEW 2

**2.1** *The GCMP generalization is not particularly novel. Putting a maximum gap size on consecutive segments is well-known in sequence mining published more than 10 years ago.*

**Response:** It is indeed that the parameter of maximum gap size has been introduced in other applications to mine the frequently occurred subsequences conforming to the gap constraint. Whereas, we are the first to introduce this parameter in trajectory co-movement mining to resolve the loose-connection anomaly. It poses new challenges and we propose two types of efficient and scalable solutions for the more generalized problem.

**2.2** *In fact, I have doubts about formulating the GCMP patterns as proposed. Are we really interested in all sets of movements beyond a cardinality of size $M$? Take the Taxi dataset as an example. Let say that there are lots of taxis going from the airport to downtown. Let say that there are 1000 such taxis. For a given $M$, are we interested in $\binom{1000}{M}$ answers? So this speaks to the problem of picking $M$. If $M$ is 500, what is $\binom{1000}{M}$? In fact, even if the system gives the single answer of $\binom{1000}{1000}$, I am not sure I am interested in this pattern as I already know that there are many taxis going from the airport to downtown. What I think I am really interested in are GCMP that are "anomalous", which is much harder to define.*

**Response:** We understand the reviewer's concern and we will respond to this issue from several aspects. First, in this example, the reviewer actually has the prior knowledge that there are lots of taxis going from airport to downtown. Hence, the results from the co-movement patterns may not look so interesting. If we consider another application in which the database contains the trajectories of all the citizens and our goal is to mine social relationships among them (i.e. friends may exhibit co-movement patterns), setting a proper $M$ becomes quite interesting and challenging. This is normally solved in an interactive way as in this case we don't have any prior knowledge. Second, it is possible that when $M$ is not set properly, an enormous number of "valid" patterns will be returned, as noticed by the reviewer. To solve this problem, a typical way is to use the concept of "closed" pattern, in which only the valid superset is returned. This can be seen as an incremental post-processing over our solution. Finally, we want to stress that we simply follow the definitions of co-movement patterns that are commonly adopted in the previous literature.

**2.3** *Regarding the second weak points, one line of related work is the superimposition of constraints on spatiotemporal mining. An example is a road network. In other words, given a road network, the network imposes constraints on GCMP.*

**Response:** We agree with the reviewer that superimposition of constraints such as road network is an interesting research direction for our future work. In this paper, our goal is to first propose a unified definition of co-movement pattern that can generalize many previous works, and then design two scalable frameworks on Spark to solve this fundamental and general problem. We believe such a work can also attract other researchers' attention to contribute and consider more advanced scenarios with superimposition of constraints on spatiotemporal mining, as suggested by the reviewer.

## 3. RESPONSES TO REVIEWER 3

**3.1:** *Though the implementation references Spark as the implementation platform (as it also reflects in github repo provided), the algorithm design is mostly limited to MapReduce, aka only Hadoop, which is a very small subset of Spark. This may have a negative impact on the baseline implementation. Particularly, recent releases of Spark have introduced window functions that can be applied directly in the sliding window scenario here. Certainly, the algorithm has to be redesigned to use DataFrame (and/or Spark SQL) interface, it has been noted that this is a very efficient way to execute window functions in Spark.*

**Response:** We thank the reviewer for the constructive comment. In our original submission, we do not implement TRPM using Spark-SQL because the latest Spark at that time (i.e., version 1.6.2 released in June) did not support *user defined aggregate function* (UDAF) on the window operator[1]. Without UDAF, we cannot integrate the Line Sweep Mining (Algorithm 1) with the window functions. During our revision period, Spark 2.0.0 is released with the support of UDAF. Thus, we explore implementing TRPM using Spark-SQL window functions.

We construct a Spark-SQL query for TRPM (referred to as TRPM-SQL) in three steps: First, we reorganize the input to form a Dataset[2] with two columns: "Time" and "Clusters". Second, we create a WindowSpec to represent the sliding window with size $\eta$ on the "Time" column. Last, we apply the WindowSpec and our Line Sweep Mining Algorithm (as UDAF) on the input Dataset. We compare the performance of TRPM-SQL with our original solutions in the following figure.

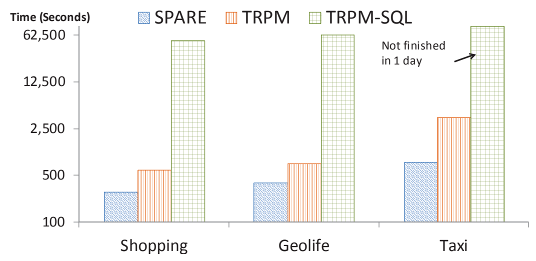

**Figure 1: Performance comparison with SPARE, TRPM and TRPM-SQL**

The figure clearly shows that TRPM-SQL is inefficient with over 120 times slower than TRPM. After our investigations on the query execution plan, we find the root cause of TRPM-SQL's slow performance is the Spark-SQL Catalyst optimizer. Since our TRPM-SQL query does not have a "partition by" clause, the Catalyst optimizer treats the entire dataset as a SINGLE partition. Then it assigns the SINGLE partition to only ONE executor, which surrenders the parallel benefits of the Spark framework. Due to the inefficiency of TRPM-SQL (in Spark version 2.0.0), we do not use it to replace the baseline (i.e., TRPM) in our revision.

**3.2:** *Most implementations on Hadoop and Spark are very sensitive to data partitions, i.e. prone to data skewing issues. It seems that the (starpartition) implementation does not take this into account, and only use the default partitioning. It would be very helpful to have a discussion on this topic.*

**Response:** In our original submission, we actually have studied the skewness issue. In Section 5.3, we propose a best-fit strategy for SPARE to achieve a better workload balance among the reducers because star partitioning may result in partitions with a different number of vertexes. In the experimental study in Section 6.2.2, we compare the SPARE algorithm with SPARE-RD and the latter only adopts the default random partition strategy. We present the summary of the execution time for each method on the longest job and standard deviation of all jobs in Table 9. The results show that SPARE demonstrates better load balance (lower deviation) than SPARE-RD, which verifies the effectiveness of the best-fit strategy.

**3.3:** *In particular, it would be great to provide the difference in the number of partitions/splits, the amount of processing and memory usage (i.e., vcore and memory seconds) between TRPM and SPARE*

**Response:** We thank the reviewer for the suggestion and we have explicitly clarified these points in the revision.

In our implementation, we take a fixed 486 partitions (Section 6) for each RDD. The partition number is determined according to the parallel tuning guide by Cloudera[3]: there are $3 \times 54 = 162$ cores and each core is able to processes 3 partitions[4] by multithreading.

Next, we add Table 7 (Section 6.1) to compare the resource usage between SPARE and TRPM in terms of Vcore-seconds, Memory and Running Time. The results show that both SPARE and TRPM are resource efficient. In particular, the RDDs for both SPARE and TRPM only take less than 20% of the available memory. This infers that SPARE and TRPM can handle even larger trajectory databases with our current cluster resources.

**3.4:** *A plot that breaks down the performance gain by each method would be greatly appreciated by the readers.*

**Response:** We thank the reviewer for the advice. In the revision, we add an analysis of the breakdown cost in Figure 8. The SPARE and TRPM algorithms have similar workflow: partitioning (Star partition v.s. $\eta$-replicate partition) in the map-shuffle phase, and mining (Apriori Enumeration v.s. Line Sweep) in the reduce phase. As shown in Figure 8, both star partitioning and apriori enumeration contribute to the final performance gain of SPARE. Specifically, in the map-shuffle phase, SPARE saves 56%-73% in time as compared to TRPM. This indicates that fewer data are trans-

[1]Window function improvements: `https://issues.apache.org/jira/browse/SPARK-7712`

[2]In Spark 2.0.0, Dataset is used to replace DataFrame.

[3]Spark Performance Tuning `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`

[4]This number may be different for other CPU models, and can be determined empirically.

formed and shuffled in the star partitioning. In the reduce phase, SPARE saves 46%-81% time as compared to TRPM. This confirms the efficiency of our Apriori enumerator.

**3.5** *Some choices of words may need to be reconsidered: for example, "a bunch of" might not be appropriate in a technical paper.*

**Response:** We have changed these unprofessional terms.

**3.6** *References to star partitioning and apriori pruning are missing. Though these are well known, they need to be clearly cited.*

**Response:** We have added the corresponding references in Section 5.1 and Section 5.2 when describing our algorithms.

**3.7** *In "In contrast, when utilizing the multicore environment, SPAREP achieves 7 times speedup and SPARES achieves 10 times speedup.", was "multicore" referring to the use of all 16 cores in one of your node? The specification of the machine was not clear.*

**Response:** We used all cores in the single node (i.e., 4 executors each with 3 cores) to conduct the experiment. We have revised the description in Section 6.2.3.

**3.8** *The computation of "eta" was slightly different than that in the paper*

**Response:** Thanks for pointing this out. We have updated the GitHub repository to rectify the typo in the equation.

## 4. RESPONSE TO META REVIEWER

**M.1** *Novelty: The GCMP generalization is not particularly novel. Please elaborate on the novelty of the work.*

**Response:** The novelty of the paper includes:

- We are the first to incorporate the parameter of maximum gap in the co-movement pattern definition.
- We are the first to deploy a distributed solution framework on Spark to solve the generalized pattern mining problem.
- There are also several novel ideas in our SPARE algorithm. For example, we devise a novel Star Partitioning to achieve better workload balance. In order to apply Apriori algorithm to overcome the exponential enumeration problem, we also devise novel concepts of sequence simplification to find a new type of monotonicity which can significantly reduce the number of candidates.

**M.2** *Parameter setting: Are we really interested in all sets of movements beyond a cardinality of size M? Please elaborate on the motivation.*

**Response:** We address this issue in Comment 2.2.

**M.3** *Spark implementation: the paper references Spark as the implementation platform but the algorithm is limited to MapReduce. Spark has capabilities beyond MapReduce such as window functions. Please provide an implementation that uses the relevant features of Spark, or a convincing discussion of how these features can be useful and why they were not used.*

**Response:** We address this issue in Comments 1.1 and 3.1.

**M.4** *More details in the performance evaluation: Please provide more details about data partitioning and the effect of skew. Also provide details about how star partitioning and a priori pruning contribute to performance. Please provide references to these two methods.*

**Response:** We address these issues in 3.2, 3.3, 3.4 and 3.6.